



ulm university universität
uulm

Correct Configuration of Process Variants in Provop

Alena Hallerbach, Thomas Bauer, Manfred Reichert

Ulmer Informatik-Berichte

**Nr. 2009-03
Februar 2009**

Correct Configuration of Process Variants in Provop

Alena Hallerbach¹, Thomas Bauer¹, and Manfred Reichert²

¹ Group Research and Advanced Engineering, Daimler AG, Ulm, Germany
{alena.hallerbach, thomas.tb.bauer}@daimler.com

² Institute of Databases and Information Systems, Ulm University, Germany
manfred.reichert@uni-ulm.de

Abstract. When engineering process-aware information systems (PAISs) one of the fundamental challenges is to cope with the variability of business processes. While some progress has been achieved regarding the configuration of process variants, there exists only little work on how to accomplish this in a correct manner. Configuring process variants constitutes a non-trivial challenge when considering the large number of process variants that exist in practice as well as the many syntactical and semantical constraints a configured process variant has to obey in a given context. In previous work we introduced the Provop approach for configuring and managing process variants. This paper picks up the Provop framework and shows how it ensures correctness of configurable process variants by construction. We discuss advanced concepts for the context- and constraint-based configuration of process variants, and show how they can be utilized to ensure correctness of the configured process variants. In this paper we also consider correctness issues in conjunction with dynamic variant re-configurations. Enhancing PAISs with the capability to correctly configure process models fitting to the given application context, and to correctly manage the resulting process variants afterwards, will enable a new quality in PAIS engineering.

Key words: process-aware information system, process variant, process configuration, correctness by construction

1 Introduction

Process support is required in almost all business domains [1]. As examples consider healthcare [2], automotive engineering [3, 4], and public administration [5]. Characteristic process examples from the automotive industry, for instance, include product change management [6] and release management [3]. When engineering process-aware information systems (PAIS) one of the fundamental challenges is to cope with business process variability and the large number of variants that may exist for a particular process [4, 7–9]. Usually, each of these variants is valid in a particular context [10].

Regarding vehicle repair in a garage, for example, we have identified hundreds of process variants which smoothly differ from each other depending on country-specific, garage-specific, and vehicle-type-specific characteristics. Similar observations can be made for release management processes, for which we identified more than 20 different variants in an automotive company depending on the respective car series, involved suppliers, and development phases [3]. Or when studying the product creation process

in the automotive domain, we can identify dozens of variants. Each of them is assigned to a particular product type (e.g., car, truck, or bus) with different organizational responsibilities and strategic goals, or varying in some other aspects. Generally, the configuration of a particular process variant depends on concrete requirements building the *process context* [10].

While some progress has been achieved regarding the modeling and management of process variants, there are only few approaches dealing with the fundamental question how to configure variants out of a master process such that correct and consistent execution behavior can be guaranteed for them [7–9, 11]. Though there exists considerable work on how to ensure structural and behavioral soundness of single process models, issues related to the correct configuration of process variants have been neglected in most cases so far. Here, the challenge is to guarantee soundness of a whole process family (i.e., a collection of process models) taking into account syntactical as well as semantical constraints to be met by the configured variants. Thereby, our goal is not to develop just another approach for checking soundness of single process models, but to provide a framework for configuring semantically valid as well as sound process variants. In particular, soundness checks should be limited to the process variants being relevant in practice, instead of considering all configurable variants. This is particularly important for scenarios in which a large number of variants exists.

In previous work we introduced the Provop (Process Variants by Options) approach for configuring and managing process variants [10]. Provop considers the whole process life cycle and supports variants in all phases following an operational approach [4]. More precisely, a concrete variant can be configured out of a master process model (denoted as *base process* in Provop) by applying a set of high-level change operations to it [12]. This paper extends the Provop framework and shows how variant configuration can be accomplished such that we obtain sound variant models afterwards. Besides correctness of variants configured at design time, we also consider correctness issues in connection with dynamic reconfiguration of process variants due to context changes.

Section 2 gives background information on Provop. Section 3 extends the Provop framework by enabling context- and constraint-based configuration of process variants. Picking up this extension, Section 4 presents an algorithm that ensures correctness of all configurable process variants by construction. We extend these considerations in Section 5 by considering dynamic variant reconfigurations as well. Finally, Section 6 discusses related work and Section 7 concludes with a summary and outlook.

2 Backgrounds - The Provop Approach

Generally, a process model variant (*process variant* for short) can be created by “cloning” a given process model and adjusting it according to the specific requirements of its application context [13]. Provop adopts this metaphor for variant creation.

Note that Provop has not been developed with a specific process formalism in mind, but shall provide an overall conceptual framework for variant modeling and management. We further assume that a process schema can be changed by applying a sequence of (high-level) change operations (e.g., to insert, delete or move process fragments) to it. We define process schema change and process variants as follows:

Definition 1 (Process Change and Process Variant)

Let \mathcal{P} denote the set of possible process schemes and \mathcal{C} the set of possible process changes. Let $S, S' \in \mathcal{P}$ be two process schemes, let $\Delta \in \mathcal{C}$ be a process change, and let $\sigma = \langle \Delta_1, \Delta_2 \dots \Delta_n \rangle \in \mathcal{C}^*$ be a sequence of process changes. Then:

- $S[\Delta]S'$ iff Δ is applicable to S and S' is the process schema resulting from the application of Δ to S .
- $S[\sigma]S'$ iff $\exists S_1, S_2, \dots, S_{n+1} \in \mathcal{P}$ with $S = S_1, S' = S_{n+1}$, and $S_i[\Delta_i]S_{i+1}$ for $i \in \{1, \dots, n\}$. We also denote S' as variant of S .

For describing process changes, Provop supports well-defined change patterns [14, 19]: *INSERT fragment*, *DELETE fragment*, *MOVE fragment*, and *MODIFY attribute*. While the first three patterns may be applied to a model fragment (i.e., a connected subgraph in Provop), the latter pattern can be used to modify the value of a process element attribute. (We provide a formal semantics of change patterns in [15]).

In Provop a process schema (denoted as *base process* is the following) can be associated with *adjustment points* that correspond to entries or exits of activities and connector nodes respectively (cf. Fig. 1 and Definition 2). This, in turn, enables designers of process-specific adaptations to refer to selected process fragments. By the use of explicit adjustment points we can restrict the regions of the base process to which adaptations (e.g., insertion or deletion of a fragment) may be applied when configuring a variant. Finally, to enable more complex process adaptations as well as their reuse in different context, Provop allows to group change operations into reusable operation sets, which we denote as *options* (cf. Definition 2). In summary, a particular variant can be configured by applying one or more options to the given base process.

Definition 2 (Base Process, Options, and Process Family)

Let \mathcal{P} be the set of process models and \mathcal{C} the set of possible process changes. Let further each process change $\Delta \in \mathcal{C}$ be represented as *parameterized change operation*. Then:

- A **base process** $S = (N, E, AP, \dots) \in \mathcal{P}$ is a process model with node set N and edge set E . Additionally, it is associated with a set of adjustment points $AP \subseteq Identifiers \times N \times \{pre, post\}$. Thereby, adjustment point $ap = (id, n, pos) \in AP$ either corresponds to the entry ($pos = pre$) or exit ($pos = post$) of node $n \in N$
- Let $S = (N, E, AP, \dots) \in \mathcal{P}$ be a base process. An associated **option** $o = (oid, \sigma)$ then corresponds to a sequence of process changes (i.e., $\sigma = \langle \Delta_1, \Delta_2 \dots \Delta_n \rangle$) that may be applied to S (or a variant derived from it). Thereby, Δ_k ($k = 1 \dots n$) refers to corresponding adjustment points or process elements (i.e., nodes and edges).
- Let $S = (N, E, AP, \dots) \in \mathcal{P}$ be a base process and let O be the set of all options defined for it. Let further $\Sigma = \langle o_1, \dots, o_n \rangle$ be a sequence of options with $\{o_1, \dots, o_n\} \subseteq O$ and $S[\Sigma]S'$. Then we denote S' as **process variant** configurable out of S by applying option sequence Σ . Further, we denote the total set of process variants that may be configured out of a given base process and an associated options set as **process family**.

Fig. 1 presents basic Provop elements along a simple example. The depicted base process represents a vehicle repair process. The process starts with the reception of a

vehicle in the garage. After a diagnosis is made, the vehicle is repaired (if necessary). The process finishes when handing over the repaired and maintained vehicle back to the customer. Depending on the process context, different variants are required. In our simplified example, three predefined options exist, out of which a subset can be chosen to configure a particular variant. Option 2, for example, suggests to insert activity *Maintenance* between adjustment points *Start Treatments* and *Treatments finished*. Option 3 itself comprises two operations which allow to insert activity *Commissioning Subcontractor* and to update attribute *Role* of activity *Maintenance*. Fig. 4 shows different variants that can be configured out of the base process from Fig. 1 by applying a subset of the defined options. Note that for more complex examples the number of variants becomes by far larger (e.g., dozens up to hundreds of variants for a vehicle repair process in automotive companies), and thus more options have to be defined to cover all cases.

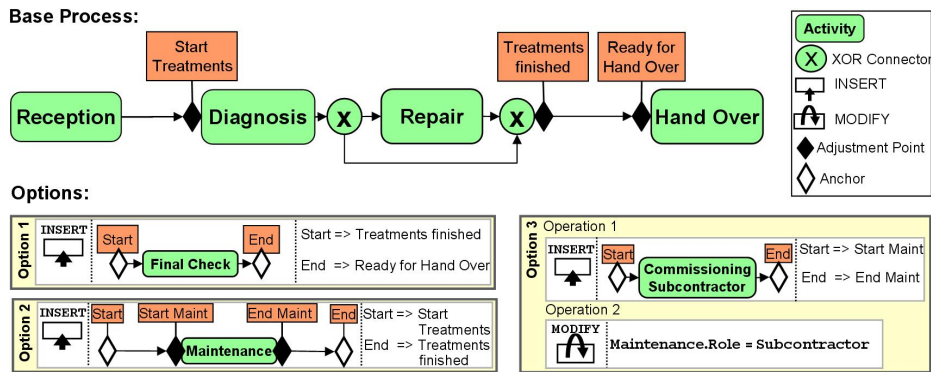


Fig. 1: ProVop example with base process and options

3 Context- and Constraint-based Variant Configuration

ProVop targets at *correctness by construction*; i.e., we want to ensure structural and behavioral soundness of all configurable process variants already at design time. As aforementioned, the focus of this paper is less on checking soundness of single process models, but more on how to reduce the number of process variant models for which soundness checks has to be checked. In addition, configuration should be accomplished automatically making use of the process context and considering semantic constraints regarding possible adaptations of the given base process.

One possible, but naive approach for guaranteeing correctness of configurable variants would be to apply all possible combinations of options to the base process and to check soundness for each of the resulting process models. However, this approach would be very expensive. As example consider the simple scenario from Fig. 1 for which three options exist. Assuming that options are not commutative in general we would have to check for 16 different option combinations whether or not their application to the base process would result in a sound process variant. Obviously, for more complex scenarios with dozens of options this is not a feasible approach.

To better understand those factors which are relevant for configuring process variants, we conducted several case studies in the automotive and the healthcare domain.

From this case study research we have learned that there is a strong linkage between the adaptations becoming necessary to configure a specific variant and the current process context; i.e., the concrete adaptations of the given base process depend on the process and application context respectively. Furthermore, we have learned that there exist different kinds of relations between the potential adaptations of a base process. While certain adaptations are mutually exclusive, others are always applied conjointly. If we explicitly express such (option) constraints we will be able to reduce the number of possible permutations and thus decrease efforts for guaranteeing soundness of configurable variants. This section summarizes how Provop enables context- and constraint-based variant configuration. We will pick up these concepts in Section 4 when presenting the Provop approach for guaranteeing soundness of configurable process variants.

3.1 Context-aware Selection of Options

As particular process variants are often required in a specific context, Provop allows for the *context-based configuration* of business process variants. For this purpose, a *context model* capturing the *process context* has to be provided. Such context model comprises a set of *context variables* as depicted in Fig. 2a. Thereby, each context variable specifies one particular dimension of the process context.¹ Regarding our vehicle repair process, for example, this can be visualized by a *context cube* as depicted in Fig. 2c. Each sub-cube then represents one possible combination of values assigned to the different context variables. We denote respective value assignments as *context descriptions* in the following. As not all possible context descriptions are semantically meaningful, Provop allows to restrict them by *context constraints* (cf. Fig. 2b). Regarding the given example, for instance, activity Maintenance will have to be performed anyway if the required security level is high. Consequently, the corresponding context constraint (cf. Fig. 2b) invalidates sub-cubes 16, 17, and 18 in Fig. 2c.

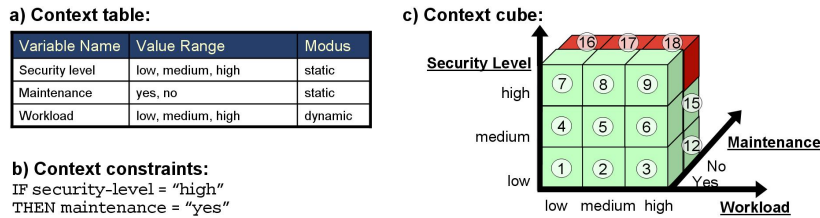


Fig. 2: Context model with corresponding table (a), constraints (b) and context cube (c)

For each process family we can define such a *context model*. Based on it *context rules* can be created and assigned to one or more *options*. This, in turn, enables context-aware option selection; i.e., if the context rule of a particular option evaluates to `true` in a given *context description* this option will be (automatically) applied to the base process when configuring a variant in the given context. Generally, for a particular *context description*, the context rules of multiple options may evaluate to `true`. In such case all

¹ In this paper we assume that context variables have a discrete and finite value range.

these options shall be considered when configuring the process variant out of the base process. (We will discuss later in which order the options shall be applied.)

Fig. 3 visualizes the previously introduced options together with their associated context rules and constraints (see below). From the context rule of Option 2, for example, we can conclude that Option 2 will be applied to the base process if context variable `Maintenance` has value “Yes” for a given context description.

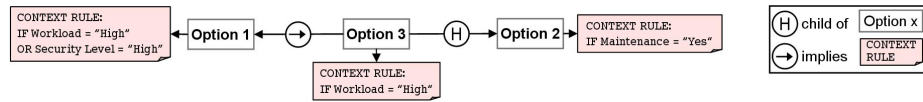


Fig. 3: Option constraints

3.2 Constraint-based Use of Options

The adjustments which become necessary to configure a particular process variant are often structurally or semantically correlated. Regarding our example from Fig. 1, for instance, the application of Option 3 to the depicted base process requires the prior introduction of Option 2 (since Operation 2 of Option 3 refers to the activity inserted by Option 2). Besides such structural dependencies semantical constraints have to be considered as well. For instance, Option 3 semantically implies Option 1 since activity `Maintenance` will always require subsequent execution of activity `Final Check`, if maintenance is not done by the garage itself, but quality of service has to be ensured. (Option 3 updates the role attribute of activity `Maintenance` to `Subcontractor`.) Protop supports three different types of option relations in order to express constraints for the use of options (see Fig. 3 for an example).

- **Implication:** If two options shall be always applied together to the base process (e.g. due to semantical dependencies) the option designer may define a directed implication relation between them. Generally, from relation “*Option 1 implies Option 2*” we must not conclude the reverse relation (i.e., *Option 2 implies Option 1*).
- **Mutual exclusion:** This constraint is useful to specify that two options must never be applied together when configuring a specific process variant.
- **Option hierarchy:** The explicit definition of an *option hierarchy* enables inheritance of change operations. If an option with ancestors is selected to configure a particular process variant, its ancestor options will be applied as well. This structures the total set of options, and also reduces the average number of change operations needed to define a particular option.

Generally, options and their change operations respectively are not commutative. Consequently, for both options and operations we need to define the order in which they shall be applied at configuration time. Based on this information, Protop allows to configure process variants through the sequential application of a set of options (and their change operations) to the given base process. In particular, the chosen option set has to match the current process context and to comply with the defined option constraints. How the latter two issues are achieved and how Protop guarantees soundness of the configurable variants is shown in the next sections. For example, Fig. 4 shows all process variants that may be derived from the base process and the options shown in Fig. 1.

Note that only those option sets are considered that match to the given context and are compliant with the defined option constraints (cf. Fig. 3).

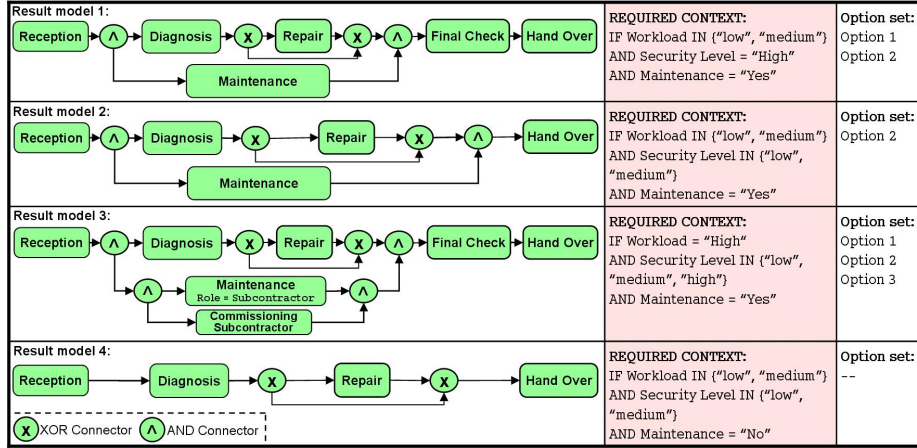


Fig. 4: Resulting process family

4 How to Guarantee Soundness of Configurable Variants

This section deals with the fundamental question how Provop ensures correct configuration of process variants without need for intense user interactions.

4.1 Basic Issues and Motivation

One possibility to ensure correctness of configured process variants is to start the configuration procedure with a sound model of the base process and to enforce soundness after each applied change operation. Consequently, the application of a set of operations and finally a set of options would result in a sound process model again. This rather rigid approach necessitates precise pre- and postconditions for the different change operations and requires sophisticated mechanisms to satisfy the pre- and post-conditions. Another possibility is to first apply the desired options to the base process and then to check soundness of the resulting process model afterwards. Provop follows the second approach since it provides more flexibility to the process designer. As pointed out in the previous section the challenge then is to reduce the number of configurable models to be checked by only considering those candidates for which the applied options satisfy the corresponding context rules and meet the defined constraints.

Provop provides high flexibility to users and supports different policies regarding the definition of the base process for a process family. For example, a base process may be designed such that it covers all configurable variants or only constitutes a minimal process model (i.e., an *intersection* of its variant models) [4]. Unlike existing configuration techniques (e.g., [7]), therefore, Provop does not necessarily require a correct process model as starting point for variant configuration. As example consider Fig. 5a where Variant 1 describes a car-specific and Variant 2 a bus-specific process variant. If

we define the base process as "intersection" of these two variant models, we obtain the process model depicted at the bottom of Fig. 5a. This base process comprises activities CA1, CA2, and CA3, but does not contain the car- or bus-specific activity. Interestingly, the shown model is not correct since the data element read by activity CA3 is neither written by CA1 nor by CA2. However, this will be not a problem if we can ensure that the variant model resulting after configuration is correct (see below).

Enforcing a correct base process is not appropriate in the given scenario. When choosing the model of Variant 1 as base process (cf. Fig. 5b), for example, visibility constraints may become violated. (Note that Variant 1 would then be visible to the process owner of Variant 2, who additionally must be able to track the adjustments of Variant 1 in order to correctly evolve Variant 2 over time). Another inadequate approach would be to restore correctness of the base process model by adding an abstract activity to the base process, which writes the data element. However, this might increase modeling efforts unnecessarily when configuring the concrete variants depicted in Fig. 5a.

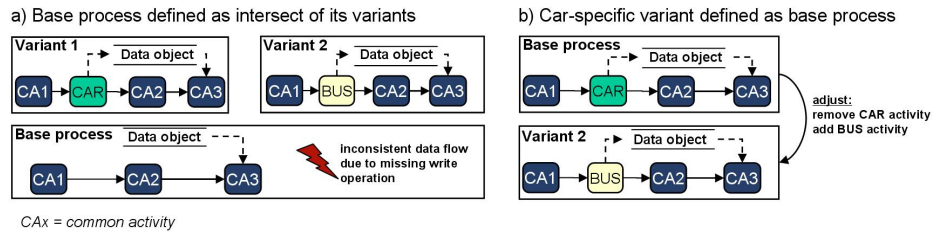


Fig. 5: Inconsistent base process (a) with an exemplary correctness scenario (b)

4.2 Overview of the Provop Correctness Checking Framework

Guaranteeing soundness of configurable process variants is accomplished in several steps (cf. Fig. 6). In Steps 1 and 2 valid context descriptions are identified, and for each of them the corresponding option set (i.e., adjustments of the base process) is determined. Step 3 then checks whether or not the calculated option sets comply with the defined option constraints (cf. Section 3). If an option set is not valid an error will be reported to the designer. Otherwise, Steps 4 and 5 apply each potential option set to the base process and check whether or not the resulting process variant model is sound. Results of Steps 4 and 5 are logged in a report (i.e., `ResultList`). In the following we describe these five steps in more detail.

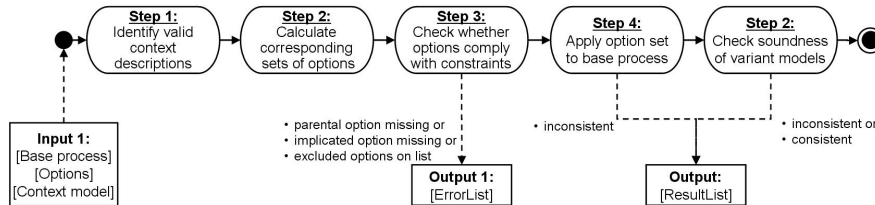


Fig. 6: Overview of the Provop procedure for guaranteeing soundness

4.3 Basic Steps of the Provop Correctness Checking Framework

We now describe the sketched procedure for checking soundness of a process family (i.e., a collection of process variants) in detail. It starts with identifying all valid context descriptions, for which process variants have to be configured. Consequently, for corresponding cases we need to guarantee correctness of the configurable variants. As invalid context descriptions are implicitly specified by the given set of context constraints, Provop first evaluates all possible allocations of values for context variables. In order to check whether or not a given context description is valid, function `contextDescriptionValid` (cf. Appendix A.5) is provided. For a given context description this function will return `true` if there is no context constraint in the given context model that invalidates this context description. As result of Step 1 we obtain the set of all valid context descriptions (stored in the variable CD_{valid}).

```

// Step 1: Identify valid context descriptions
CDvalid = ∅ // Initializing the set of valid context descriptions
// create context descriptions by simulating all possible values in the value range of each
// context variable CtxtVari, i=1,.., n defined in the context model. We assume that corre-
// sponding value ranges ValueRange(CtxtVari) are discrete and finite.
for all CtxtDescription ∈ (ValueRange(CtxtVar1) × ... × ValueRange(CtxtVarn)) do
  // check whether or not the current context description is valid
  if contextDescriptionValid(CtxtDescription) = true then
    CDvalid := CDvalid ∪ {CtxtDescription}

```

Example 1: In our example from Fig. 2 context cubes 16, 17, and 18 become invalid due to the context constraint "IF Security-level = high THEN Maintenance = yes". All other context descriptions are valid. Therefore, we add them to set CD_{valid} of valid context descriptions; i.e., $CD_{valid} = \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$.

For each valid context description Step 2 calculates the option set to be applied when configuring the corresponding variant. (An option set can be empty as the base process itself can be a variant.) For this purpose, Step 2 utilizes function `contextRuleValid` (cf. Appendix A.6), which can be used to check whether or not a single option shall be applied in the given context. This function will return `true` if the context rule of an option (cf. Sect. 3) is valid regarding currently chosen values of the context variables (i.e., regarding the given context description). Thus `contextRuleValid` is applied to each option. If it returns `true` for a selected option, this option will be added to the option set of the currently considered context description. Otherwise it will be not considered in the given context.

As depicted in Fig. 7 different context descriptions may have same option set. To check only once whether or not a particular option set is applicable to the base process, context descriptions with same option set are grouped into *context blocks*. This is accomplished by function `extendCtxtBlock`. As result of Step 2 we obtain a set of $\langle \text{ctxtblock}, \text{option set} \rangle$ pairs; i.e., for each context block (i.e., set of context descrip-

tions) we obtain the option set to be applied for variant configuration when corresponding context becomes valid – we denote this object as *process variant candidates*.

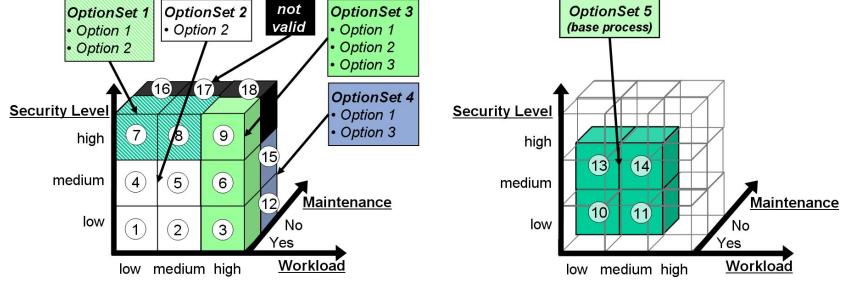


Fig. 7: Blocks of valid context descriptions and respective sets of options

```

// Step 2: Calculate corresponding sets of options
// consider set of valid context descriptions  $CD_{valid}$  as determined in Step 1
for each CtxtDescription  $\in CD_{valid}$  do
    CalculatedOptions :=  $\emptyset$ 
    // check validity of context rules for all defined options
    for each Option  $\in$  allDefinedOptions do
        if contextRuleValid(Option,CtxtDescription) = true then
            CalculatedOptions := CalculatedOptions  $\cup$  {Option}
    // check if set of options has already been created
    if hasOptionSet(CalculatedOptions,ProcessVariantCandidates) = false then
        // insert new block of context descriptions with one common option set
        insertCtxtBlock(ProcessVariantCandidates,CtxtDescription,CalculatedOptions)
    else // extend existing block with current context description
        extendCtxtBlock(ProcessVariantCandidates,CtxtDescription,CalculatedOptions)

```

Example 2: For the context model from Fig. 2 we obtain the following process variant candidates (i.e., $\langle \text{CtxtBlock}, \text{OptionSet} \rangle$): $\langle \{12, 15\}, \{\text{Option 1}, \text{Option 3}\} \rangle$, $\langle \{7, 8\}, \{\text{Option 1}, \text{Option 2}\} \rangle$, $\langle \{1, 2, 4, 5\}, \{\text{Option 2}\} \rangle$, $\langle \{3, 6, 9\}, \{\text{Option 1}, \text{Option 2}, \text{Option 3}\} \rangle$, $\langle \{10, 11, 13, 14\}, \{\} \rangle$. The option set of the latter candidate is empty; i.e., its model corresponds to the base process.

Step 3 checks for each process variant candidate the semantic compatibility of its option set with the defined constraints. This could be based, for example, on LTL or some other model checking technique. Here, we simply assume that there is a function `checkOptionConstraints`, which will return `true` if the corresponding option set complies with all explicitly defined option constraints (cf. Section 2). Otherwise, the respective $\langle \text{CtxtBlock}, \text{OptionSet} \rangle$ pair is deleted from the set of process variant candidates and corresponding information is added to an error report (i.e., `ErrorList`).

```

// Step 3: Check whether options comply with constraints
for each <CtxtBlock,OptionSet> ∈ ProcessVariantCandidates do
  if checkOptionConstraints(OptionSet) = false then
    // remove candidates that are not complaint with option constraints
    removeCandidate(ProcessVariantCandidates,<CtxtBlock,OptionsSet>)
    insertInErrorList(..) // write entry to an error report, including in compliant OptionSet

```

Example 3: The hierarchy constraint described in Section 3 requires that all ancestors of an option are also applied to the base process. As example consider the constraints defined for the options from Fig. 3. OptionSet 4, which contains Options 1 and 3 (cf. Fig. 7), does not comply with the hierarchy constraint. Reason is that ancestor of Option 3 (i.e., Option 2) is not contained in the option set. Consequently, the context block associated with OptionSet 4 is removed from the list of variant candidates. Generally, in addition to context-dependencies, option constraints ensure semantical correctness of option sets and further reduce efforts for checking correctness.

After completing Step 3 we have identified relevant process variant candidates. For each candidate the elements from its option set have to be ordered, i.e. the sequence in which the options shall be applied has to be fixed. Provop provides different concepts for ordering options (e.g., based on time stamps or user defined order). Due to lack of space we omit details here, but assume that there is a function `sortedOptionSet` (cf. Appendix A.10) that defines a partial order for options. If an error occurs, e.g. due to cyclic ordering constraints explicitly defined by the user, the procedure will stop and add an entry to the report list, which specifies that the current process variant candidate has turned out to be inconsistent.

```

// Steps 4+5: Apply option set to base process and check soundness of variant models
for each <CtxtBlock,OptionSet> ∈ ProcessVariantCandidates do
  // create partial order of OptionSet in sortedOptionList
  if sortOptionSet(OptionSet,sortedOptionList) = true then
    // calculate variant model by applying an option set to the base process
    if calculateVariant(BaseProcess,sortedOptionList,VariantModel) = true then
      if checkSoundness(VariantModel) = true then // variant model is sound
        storeResult(OptionSet, "sound", ..)
      else // variant model is not sound
        storeResult(OptionSet, "not sound", ..)

```

After successfully executing Step 4, for each process variant candidate we have obtained its option set and the order in which the options shall be applied to the base process when configuring the corresponding variant. Step 5 then calculates the candidate models, if possible, by using function `calculateVariant` (cf. Appendix A.2). If an option and its associated change operations are not applicable (e.g., due to missing object references) Provop will not calculate a candidate model for the corresponding option set, but will add an entry to an error report. Otherwise, structural and behavioral

soundness of the resulting variant model are checked, considering the specifics of the underlying meta-model (function `checkSoundness` in Appendix A.4). Finally, the process variant candidate, together with the respective consistency check result (i.e., either “consistent” or “inconsistent”) are stored in the report list.

5 Dynamic Reconfiguration of Process Variants

In a dynamic environment it often becomes necessary to adjust running processes [13]. Provop captures this by enabling reactions on context changes and by dynamically re-configuring variants; i.e., to switch from one variant model to another during runtime.

5.1 Motivation and Basic Issues

Basically, we distinguish between static and dynamic context variables. A *static context variable* is set once at configuration time and is not supposed to change during variant execution. Opposed to this, a *dynamic context variable* may be updated during runtime. For example, the context model of a vehicle repair process contains context variable `Security-level`, which is defined as static as no value updates occur during variant execution (cf. Fig. 2). Context variable `Workload`, in turn, is updated from time to time according to the current workload level of a specific garage. Consequently, this context variable is defined as dynamic. The challenge now is to cope with such dynamic (i.e., changing) process context and to be able to switch to another process variant on-the-fly.

If the context rule of an option refers to a dynamic context variable the decision whether or not to apply the corresponding change operations has to be deferred to runtime. Consequently, the referred variant model has to be dynamically reconfigured. Provop supports controlled use of such *dynamic options* through *variant branches* that encapsulate single change operations of a (dynamic) option. Their split condition corresponds to the context rule of the option. When a variant branch is reached at runtime, Provop evaluates the process context. If the split condition evaluates to `true` the variant branch is executed, i.e., its change operation is applied to the base process. Otherwise, the variant branch is skipped and all adjustments defined by the option are ignored.

Provop is able to cope with options comprising multiple change operations. It treats the operations of an option atomically, as this corresponds to the intention of the option designer. For instance, in our example from Fig. 1, Option 3 comprises two operations that must not be treated in isolated fashion. Assume that Operation 1 is applied due to “high” `Workload` and Operation 2 is omitted because the current context changes in the meanwhile (e.g., value of context variable `Workload` changes from `high` to `low`). Then a subcontractor is commissioned, but work still has to be done by the garage itself as the attribute `role` of activity `Maintenance` is not modified due to omitting Operation 2 of Option 3. Obviously, this is no appropriate execution behaviour. Consequently, if the first variant branch of a dynamic option is applied to the base process, all other variant branches corresponding to the same option (i.e., change operations associated with this option) will have to be applied as well.

For variant branches and the dynamic application of options we also have to ensure compliance with the defined option constraints.² Therefore, Provop makes the use of dynamic options dependent on these constraints and not only on the context rules assigned. Furthermore, option constraints are treated with higher priority than context rules. For the three options introduced in Fig. 1, Fig. 8 gives an example of a dynamic reconfiguration. Based on the initial context description "Workload = medium AND Security-Level = low AND Maintenance = Yes" Option 2 is applied to the base process (cf. Fig. 8a). Assume that during execution of the configured variant, dynamic context variable Workload is updated from medium to high. Thus, context rules of both Option 3 and Option 1 (cf. Fig. 3) become valid and are added to the option set (cf. Fig. 8b). Assume further that later on another context change occurs after Option 3 has become effective (i.e., the corresponding variant branch was executed), but before the variant branch of Option 1 is reached (e.g. updating the dynamic context variable Workload from high to medium). Consequently, context rules of Options 3 and 1 become invalid. One possibility is to not apply Option 1 though Option 3 (i.e., its variant branch) has already been executed (cf. Fig. 8b). As the final check is still required for the given scenario the effects of Option 1 will not be omitted anymore. Here the implication constraint defined between Options 3 and 1 ensures that Option 1 cannot be omitted; i.e., the option set shown in Fig. 8c is not allowed at runtime.

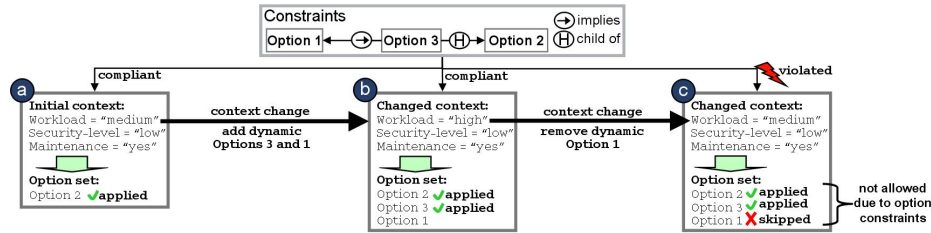


Fig. 8: Dynamic context change at runtime

Another challenge is to still ensure correct execution behavior of the variant model even if its context and thus the set of applied dynamic options varies during runtime. One rigid policy would be to ignore dynamic context changes. Obviously, this disallows reactions on context changes and dynamic variant reconfigurations (by applying or omitting options). However, dynamic checks during runtime do not always constitute a realistic solution, particularly if complex reconfigurations become necessary and end users shall be not involved. Provop ensures soundness of process variant models by checking soundness of all producible process models already at buildtime including dynamic reconfigurations; i.e., Provop checks soundness of all possible variant models that can result from dynamic reconfigurations due to context changes (cf. Fig. 8). Again the applied options must comply with the defined constraints. By preventively checking soundness for dynamically (re-)configurable variants we guarantee that runtime context changes and variant reconfigurations lead again to a sound variant model.

² In Provop a distinction is made for each option constraint whether it is only relevant in static context or will be considered at runtime as well.

5.2 Checking Soundness of Configurable Variant Models A-priori

The procedure introduced in Sect. 4 has not considered dynamic variant reconfiguration so far; i.e., dynamic context changes and changing option sets have been factored out. We now extend our method with Step 6 which also checks soundness for all process variants that may be dynamically (re-)configured. Basic to Step 6 is the idea to test for each valid context description whether or not we need a preventive soundness check that considers potential dynamic changes of context variables. This will be the case if dynamic options are added or removed from a given option set (due to its context rule becoming valid or invalid when context changes). First, we identify both static options (i.e., options not affected by context changes) and dynamic ones using function `calculateDynamicOptions` (cf. Appendix A.1). Basically, for a considered context description, this function fixes the values of static context variables, whereas it tests all possible value allocations for dynamic context variables; i.e., runtime change of dynamic variables is simulated leading to additional context descriptions. Out of this we obtain the dynamic options (`dynamicOptions`) and the static ones (`staticOptions`) by checking whether the context rule of an option is valid in all created context descriptions (i.e., the option is static), is not valid in any created context description (i.e., is not relevant for this context description at all and we can ignore it), or is valid in at least one but not all context descriptions (i.e., the option is dynamic).

```

// Step 6: Simulate dynamic context change
// use CDvalid created in Step 1 (cf. Section 4.3)
for each CtxtDescription ∈ CDvalid do
  // identify static and dynamic options
  calculateDynamicOptions(CtxtDescription, dynamicOptions, staticOptions)
  if dynamicOptions ≠ ∅ then
    // create power set of all defined options
    for each dynOptionSet ∈ getPowerSet(dynamicOptions) do
      // join subset of dynamic options with static options
      simOptionSet := dynOptionSet ∪ staticOptions
      // check if option set is compliant with defined option constraints
      if checkOptionConstraints(simOptionSet) = true then // cf. Appendix A.3
        // check entries of result list if option set has already been checked
        if simOptionSet ∈ ResultList then
          if (getResult(ResultList, simOptionSet) = "not sound") then
            insertInErrorList(...) // option set is inconsistent
          else // option set is compliant with constraints but has not been checked yet
            repeat Step 4 and 5 of the main algorithm (cf. Section 4.3)
          else insertInErrorList(..) // cf. Step 3

```

Example 4: Assume that we invoke function `calculateDynamicOptions` with context description "Workload = high, Security-level = medium, Maintenance = yes". Then, every possible value of dynamic context variable `Workload` is simulated resulting in two additional

context descriptions to be checked; i.e. for `Workload = low` and `Workload = medium` with values of static context variables `Security-level` and `Maintenance` being fixed. Following this, validity of context rules assigned to the options within these three context descriptions is checked. We obtain: `staticOptions = {Option 2}`, `dynamicOptions = {Option 1, Option 3}`.

If set `dynamicOption` created by `calculateDynamicOptions` is not empty, a dynamic reconfiguration might become necessary. Thus, we need to calculate new option sets for the context descriptions which result when considering any possible context change at any point in time during execution. First of all, static options are fixed; i.e., they are considered to be always part of each newly created option set. To simulate reconfiguration, the static options are combined with every possible subset of dynamic options; i.e., each element of the power set of `dynamicOptions` is joined with fixed set `staticOptions` resulting in a temporary option set `simOptionSet`.

Example 5: For our example the power set of `dynamicOptions` is \emptyset , `{Option 1}`, `{Option 3}`, and `{Option 1, Option 3}`. Each of these subsets is joined with `staticOptions` leading to 4 option sets: `{Option 2}`, `{Option 1, Option 2}`, `{Option 3, Option 2}`, and `{Option 1, Option 3, Option 2}`. Each of these option sets is then checked whether or not it is compliant with defined option constraints. In our example the simulated set `{Option 3, Option 2}` is not compliant. Thus, an entry to the error list is written.

After creating such temporary option set `simOptionSet` its compliance with the option constraints is checked using function `checkOptionConstraints`. We then validate whether consistency of `simOptionSet` has already been checked; i.e., applicability of options is guaranteed and the resulting variant model is sound. If we have not yet checked the option set, Steps 4 and 5 of our procedure are re-applied. Altogether, Protop enables correctness by construction for variants based on static as well as dynamic options.

6 Related Work

Though the adequate support of process variants is highly relevant for practice, only few approaches for variant management exist.

There exists adaptive process management technology that enables dynamic process changes during runtime; i.e., authorized users may dynamically adapt the structure (i.e., the schema) of running process instances (e.g., by adding, deleting or moving activities) [16, 17, 13, 18, 19]. Obviously, this runtime flexibility results in a multitude of process variants, of which each represents one particular case (i.e., process instance). The approach described in [17] additionally provides support for the management and retrieval of the resulting process instance variants. In particular, it becomes possible to store, manage, and query large collections of process variants within a process repository. Graph-based search techniques are used in order to retrieve process variants that are similar to a user-defined process fragment (i.e., the query is represented as graph). Obviously, this approach requires profound knowledge about the structure of stored process instances, an assumption which does not always hold in practice. Variant search based on process meta data (e.g., the process context) is not considered.

An important area related to variant management is reference process modeling. A reference process has recommending character, covers a family of process models, and can be customized to meet specific needs. Configurable event process chains (C-EPCs), for example, provide support for both the specification and customization of reference process models [8, 9]. When modeling a reference process, EPC functions (and decision nodes) can be annotated to indicate whether or not they are mandatory or optional. This information is considered when configuring C-EPCs. A similar approach is presented in [11]. Here the concepts for configuring a reference process model (i.e., to enable, hide or block a configurable process element) are transferred to workflow models. Similar to Provop constraints regarding the application of different adjustments of the reference process can be defined (e.g., two activities either may have to be deleted together or none of them). As opposed to Provop, it neither is allowed to move or add model elements nor to adapt element attributes when configuring a variant. Finally, [7] shows how to configure reference process models incrementally and in a way that ensures the correctness of the process variants, both with respect to syntax and behavioral semantics. As opposed to Provop, this approach assumes that the reference process model is sound.

Different work exists on how specialization can be applied to deal with process model variability taking advantage of the generative nature of a specialization hierarchy [20, 21]. [20] has shown how specialization can be realized for state and dataflow diagrams respectively. For both diagram types a set of transformation rules is provided resulting in process specializations when applying them to a particular model. Similarly, [21] discusses transformation rules to define specialization for models based on Petri Nets. Basically, specialization allows to capture process variants. As opposed to these approaches, Provop follows an operational approach, which is independent of the underlying process meta model. In addition, we provide comprehensive support for the context- and constraint-based configuration of process variants.

Fundamental characteristics of software variability in software engineering are described in [22]. In particular, software variants exist in software architectures and software product lines [23, 24]. Often feature diagrams are used for modeling software systems with varying features; correctness issues are not considered. Another contribution stems from PESOA [25] which provides basic concepts for variant modeling based on UML. Different variability techniques like inheritance, parameterization, and extension points are provided. As opposed to PESOA, Provop provides a more powerful instrument for describing variance in a uniform and easy manner. Finally, [26] goes beyond control flow and extends business process configuration to roles and objects.

7 Summary and Outlook

We have described the Provop approach for configuring and managing process variants. In this paper, we put emphasis on how to ensure correctness of configured process variants by construction, taking into account semantical as well as structural constraints. Furthermore, we considered issues related to the dynamic re-configuration of process variants due to changing process context. We have prototypically implemented

the Provop approach on top of the ARIS tool utilizing the programming interface provided by it [27]. In future research we will apply Provop in industrial context.

References

1. Mutschler, B., Reichert, M., Bumiller, J.: Unleashing the effectiveness of process-oriented information systems: Problem analysis, critical success factors and implications. *IEEE Transactions on Systems, Man, and Cybernetics (Part C)* **38** (2008) 280–291
2. Lenz, R., Reichert, M.: IT Support for Healthcare Processes - Premises, Challenges, Perspectives. *Data and Knowledge Engineering* **61** (2007) 39–58
3. Müller, D., Herbst, J., Hammori, M., Reichert, M.: IT Support for Release Management Processes in the Automotive Industry. In: *Proc. 4th Int. Conf. on Business Process Management*. (2006) 368–377
4. Hallerbach, A., Bauer, T., Reichert, M.: Managing Process Variants in the Process Life Cycle. In: *Proc. 10th Int. Conf. on Enterprise Information Systems*. (2008)
5. Becker, J., Lis, L., Pfeiffer, D., Räckers, M.: A process modeling language for the public sector - the picture approach. *Wybrane Problemy Elektronicznej Gospodarki* (2007) 271–281
6. VDA Recommendation 4965 T1: Engineering Change Management (ECM) - Part 1: Engineering Change Request (ECR) Version 1.1 (2005)
7. van der Aalst, W.M.P., Dumas, M., Gottschalk, F., ter Hofstede, A.H.M., la Rosa, M., Mendling, J.: Correctness-preserving configuration of business process models. *Fundamental Approaches to Software Engineering* (2008) 46–61
8. Rosemann, M., van der Aalst, W.: A Configurable Reference Modeling Language. *Information Systems* **32** (2007) 1–23
9. Rosa, M.L., Lux, J., Seidel, S., Dumas, M., ter Hofstede, A.H.M.: Questionnaire-driven Configuration of Reference Process Models. In: *Proc. CAiSE'07*. (2007)
10. Hallerbach, A., Bauer, T., Reichert, M.: Context-based configuration of process variants. In: *Proc. TCoB 2008 Workshop*. (2008)
11. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., la Rosa, M.: Configurable Workflow Models. In: *Int. Journal of Cooperative Information Systems*. (2007)
12. Hallerbach, A., Bauer, T., Reichert, M.: Issues in modeling process variants with provop. (In: *Proc. BPM'08 Workshops*)
13. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems – a survey. *Data and Knowledge Engineering* **50** (2004) 9–34
14. Weber, B., Rinderle, S.B., Reichert, M.: Change patterns and change support features in process-aware information systems. In: *Proc. CAiSE'07, Trondheim, Norway*. Volume 4495 of *Lecture Notes in Computer Science (LNCS)*, London, Springer Verlag (2007) 574–588
15. Rinderle-Ma, S., Reichert, M., Weber, B.: On the formal semantics of change patterns in process-aware information systems. In: *Proc. ER'08*. LNCS 5231 (2008) 279–293
16. Reichert, M., Rinderle, S., Kreher, U., Dadam, P.: Adaptive process management with adept2. In: *Proc. ICDE'05*. (2005) 1113–1114
17. Lu, R., Sadiq, S.: On Managing Process Variants as an Information Resource. Technical Report No. 464, School of Information Technology & Electrical Engineering and University of Queensland, Brisbane (2006)
18. Rinderle, S., Reichert, M., Dadam, P.: Disjoint and overlapping process changes: Challenges, solutions, applications. In: *CoopIS'04*. LNCS 3290 (2004) 101–120
19. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering* **66** (2008) 438–466

20. Malone, T., Crowston, K., Herman, G.: Organizing Business Knowledge - The MIT Process Handbook. MIT Press (2007)
21. van der Aalst, W.M.P., Basten, T.: Inheritance of Workflows: An Approach to Tackling Problems Related to Change. Technical report, TU Eindhoven (2002)
22. Bachmann, F., Bass, L.: Managing Variability in Software Architectures. In: Proc. of 2001 Symp. on Software Reusability, New York, ACM Press (2001) 126–132
23. Becker, M., Geyer, L., Gilbert, A., Becker, K.: Comprehensive Variability Modeling to Facilitate Efficient Variability Treatment. In: 4th Workshop on Product Family Eng. (2001)
24. Halmans, G., Pohl, K.: Communicating the Variability of a Software-Product Family to Customers. *Software and System Modeling* **2** (2003) 15–36
25. Puhlmann, F., Schnieders, A., Weiland, J., Weske, M.: PESOA - Variability Mechanisms for Process Models. Technical Report 17/2005, Hasso-Plattner-Institut, Potsdam (2005)
26. La Rosa, M., Dumas, M., ter Hofstede, A., Mendling, J., Gottschalk, F.: Beyond control-flow: Extending business process configuration to roles and objects. In: Proc. ER'08. (2008)
27. IDS Scheer AG: ARIS Platform Method 7.1. (2008) www.ids-scheer.com.

A Appendix

In the following we describe the main functions used by our consistency checking procedure as introduced in Sections 4.3 and 5. Each function is defined by a name, input and output parameters, and results. Furthermore, we provide a short description and pseudocode. The functions are listed in alphabetical ordering.

Global variables of the functions are the context model (i.e., `CtxtModel`) and the defined options (i.e., `allDefinedOptions`).

A.1 `calculateDynamicOptions`

`calculateDynamicOptions`

Input:

`CtxtDescription`: allocation of current values to context variables

Output:

`dynamicOptions`: option set

`staticOptions`: option set

This function identifies all static as well as dynamic options for a considered context description by fixing values of static context variables and by permuting all possible value allocations for dynamic context variables. It further checks whether or not the context rule of an option is valid for all created context descriptions (i.e., the option is static), is not valid in any created context description (i.e., the option is not relevant in this context description at all), or is valid in at least one but not all context descriptions (i.e., the option is dynamic).

```
// get all context variables defined as static in the corresponding context model
Vstat = {StatV1, .. , StatVn} ∈ CtxtModel
  with StatVi.mode = static for i = 1..n
// get all context variables defined as dynamic in the corresponding context model
Vdyn = {DynV1, .. , DynVm} ∈ CtxtModel
  with DynVj.mode = dynamic for j = 1..m

// initialization
staticOptions = ∅
dynamicOptions = ∅

// init for each option the attributes alwaysUsed and neverUsed
for each option ∈ allDefinedOptions do
  alwaysUsed(option) = true
  neverUsed(option) = true
```

```

// for each static context variable create a tuple with name and current value
// (given by current context description)
for each i = 1..n do
    StatSimVari := <StatVi.name, getValue(StatVi.name,CtxtDescription)>

// for each dynamic context variable create a set of tuples with each tuple
// compraising the name and one possible value out of the value range of a context
// variable
for each j = 1..m do
    DynSimVarj := {<DynVj.name, value1>, ..., <DynVj.name, valuek>}
    with value1 ... valuek ∈ ValueRange(DynVj)

// simulate dynamic change of context by creating a set of context descriptions out
// of the cross product of the above created tuples (i.e., <CtxtVariable.name,value>)
for each CtxtDescr
    ∈ {StatSimVar1} × ... × {StatSimVark} × {DynSimVar1} × ... × {DynSimVarm}
    for each Option ∈ allDefinedOptions do
        if contextRuleValid(Option,CtxtDescr) then
            neverUsed(Option) = false
        else alwaysUsed(Option) = false

// static options are always applied, whereas dynamic ones are applied in at least
// one but not all context descriptions
for each Option ∈ allDefinedOptions do
    if alwaysUsed(Option) then staticOptions := staticOptions ∪ {Option}
    else if alwaysUsed(Option)= false AND neverUsed(Option) = false then
        dynamicOptions := dynamicOptions ∪ {Option}

```

A.2 calculateVariant

calculateVariant

Input:

BaseProcess: base process to be transformed
sortedOptionList: sorted list of options

Output:

VariantModel: model of a specific process variant

Result:

Boolean

The result of this function will be true, if no error occurs, otherwise it will be false.

For each option from `sortedOptionList` all assigned operations are applied to the base process. Thereby, the different operation types are considered and corresponding functions are applied. The specific algorithms of our change operations are out of scope of this paper and are omitted here.

A.3 `checkOptionConstraints`

`checkOptionConstraints`

Input:

`OptionSet`: set of options

Result

Boolean

The result of this function will be `true` if all defined option constraints are met by the option set `OptionSet`.

A.4 `checkSoundness`

`checkSoundness`

Input:

`VariantModel`: model of a specific process variant

Result

Boolean

The result of this function will be `true` if the `VariantModel` is sound considering the specific soundness criteria of the underlying process meta model. Otherwise, the result will be `false` (i.e. at least one criterion is violated).

A.5 `contextDescriptionValid`

`contextDescriptionValid`

Input:

`CtxtDescription`: allocation of current values to context variables

Result

Boolean

The result of this function will be `true` if the context description is valid regarding all given context constraints of the underlying context model `CtxtModel`. Otherwise the result will be `false`. The trivial algorithm behind this function is omitted here.

A.6 `contextRuleValid`

`contextRuleValid`

Input:

Option: option defined for the current base process

CtxtDescription : allocation of current values to context variables

Result

Boolean

The result of this function will be `true` if the associated context rule to an option is valid. Otherwise the result is `false`. The apparently trivial algorithm of the function is omitted here.

A.7 `extendCtxtBlock`

`extendCtxtBlock`

Input:

CtxtDescription: allocation of current values to context variables

ProcessVariantCandidates: set of `<CtxtBlock,OptionSet>` pairs

CalculatedOptions: set of options

This function identifies a `<CtxtBlock,OptionSet>` pair within the `ProcessVariantCandidates` whose `OptionSet` is equal to option set `CalculatedOptions`. The set of context descriptions (i.e. `CtxtBlock`), of the identified pair is then extended by context description `CtxtDescription`.

A.8 hasOptionSet

hasOptionSet

Input:

CalculatedOptions: set of options

ProcessVariantCandidates: set of <CtxtBlock,OptionSet> pairs

Result

Boolean

The result of this function will be true if the option set CalculatedOptions is covered by a <CtxtBlock,OptionSet> pair from ProcessVariantCandidates. Otherwise the result will be false.

A.9 insertCtxtBlock

insertCtxtBlock

Input:

ProcessVariantCandidates: set of <CtxtBlock,OptionSet> pairs

CtxtDescription: allocation of current values to context variables

CalculatedOptions: set of options

The function creates a new context block CtxtBlock that comprises the context description CtxtDescription. Furthermore, the CtxtBlock and the option set CalculatedOptions are added to object ProcessVariantCandidates; i.e., pair <CtxtBlock,CalculatedOptions> is inserted.

A.10 sortOptionSet

sortOptionSet

Input:

OptionSet: set of options

Output:

sortedOptionList: sorted list of options

Result:

Boolean

Let $getTimeStamp(Option)$ be the function that returns the creation time of an option and let $SeqConstraint(Option_i, Option_j)$ be defined as sequencing constraint between options $Option_i$ and $Option_j$ (i.e. $Option_i$ shall be applied to the base process before $Option_j$). Let further $SeqConstraint^*(Option_i, Option_j)$ be the transitive closure of $SeqConstraint(Option_i, Option_j)$.

```

if  $\exists$  SeqConstraint*(Optioni, Optionj)
AND  $\exists$  SeqConstraint*(Optionj, Optioni) = true then // i.e. cyclic constraints
    // write an entry to the error log, including the error causing options
    (i.e., the options that are part of the cyclic sequencing constraints)
    insertInErrorList(...)
    return false
else
    for each  $i < j \in (1, \dots, n)$  do
        if  $\exists$  SeqConstraint*(Optioni, Optionj)
        OR ( $\nexists$  SeqConstraint*(Optioni, Optionj)
        AND  $\nexists$  SeqConstraint*(Optionj, Optioni))
        AND (getTimeStamp(Optioni) < getTimeStamp(Optionj)) = true then
            // sort options by any sorting algorithm
            sortedOptionList = (Option1, ..., Optionn) with Optioni < Optionj
    return true

```

Liste der bisher erschienenen Ulmer Informatik-Berichte
Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm
Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

- 91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity
- 91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming
- 91-03* *Alfons Geser*
Relative Termination
- 91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP
- 91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions
- 91-06* *Uwe Schöning*
Recent Highlights in Structural Complexity Theory
- 91-07* *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit
- 91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk, A. Ogiwara,*
U. Schöning, R. Silvestri, T. Thierauf
Reductions for Sets of Low Information Content
- 92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets
- 92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars
- 92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen
- 92-04* *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions
- 92-05* *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy
- 92-06* *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics
- 92-07* *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

- 92-08* *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets
- 92-09* *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network
- 92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function
- 92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes
- 92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets
- 92-13 *Alfons Geser*
On a monotonic semantic path ordering
- 92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers
- 93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager
- 93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model
- 93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets
- 93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions
- 93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System
- 93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung
- 93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects
- 94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights
- 94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers
- 94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP
- 94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers

- 94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems
- 94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model
- 94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages
- 94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge
- 94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software
- 94-10 *Axel Dold*
Formalisierung schematischer Algorithmen
- 94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits
- 94-12 *Rainer Schuler*
On Average Polynomial Time
- 94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems
- 94-14 *Wolfram Schulte, Ton Vullingsh*
Linking Reactive Software to the X-Window System
- 94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen
- 94-16 *Robert Regn*
Verteilte Unix-Betriebssysteme
- 94-17 *Helmuth Partsch*
Again on Recognition and Parsing of Context-Free Grammars:
Two Exercises in Transformational Programming
- 94-18 *Helmuth Partsch*
Transformational Development of Data-Parallel Algorithms: an Example
- 95-01 *Oleg Verbitsky*
On the Largest Common Subgraph Problem
- 95-02 *Uwe Schöning*
Complexity of Presburger Arithmetic with Fixed Quantifier Dimension
- 95-03 *Harry Buhrman, Thomas Thierauf*
The Complexity of Generating and Checking Proofs of Membership
- 95-04 *Rainer Schuler, Tomoyuki Yamakami*
Structural Average Case Complexity
- 95-05 *Klaus Achatz, Wolfram Schulte*
Architecture Independent Massive Parallelization of Divide-And-Conquer Algorithms

- 95-06 *Christoph Karg, Rainer Schuler*
Structure in Average Case Complexity
- 95-07 *P. Dadam, K. Kuhn, M. Reichert, T. Beuter, M. Nathe*
ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen
- 95-08 *Jürgen Kehrer, Peter Schulthess*
Aufbereitung von gescannten Röntgenbildern zur filmlosen Diagnostik
- 95-09 *Hans-Jörg Burtschick, Wolfgang Lindner*
On Sets Turing Reducible to P-Selective Sets
- 95-10 *Boris Hartmann*
Berücksichtigung lokaler Randbedingung bei globaler Zielloptimierung mit neuronalen Netzen am Beispiel Truck Backer-Upper
- 95-12 *Klaus Achatz, Wolfram Schulte*
Massive Parallelization of Divide-and-Conquer Algorithms over Powerlists
- 95-13 *Andrea Mößle, Heiko Vogler*
Efficient Call-by-value Evaluation Strategy of Primitive Recursive Program Schemes
- 95-14 *Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
A Generic Specification for Verifying Peephole Optimizations
- 96-01 *Ercüment Canver, Jan-Tecker Gayen, Adam Moik*
Formale Entwicklung der Steuerungssoftware für eine elektrisch ortsbediente Weiche mit VSE
- 96-02 *Bernhard Nebel*
Solving Hard Qualitative Temporal Reasoning Problems: Evaluating the Efficiency of Using the ORD-Horn Class
- 96-03 *Ton Vullingsh, Wolfram Schulte, Thilo Schwinn*
An Introduction to TkGofer
- 96-04 *Thomas Beuter, Peter Dadam*
Anwendungsspezifische Anforderungen an Workflow-Mangement-Systeme am Beispiel der Domäne Concurrent-Engineering
- 96-05 *Gerhard Schellhorn, Wolfgang Ahrendt*
Verification of a Prolog Compiler - First Steps with KIV
- 96-06 *Manindra Agrawal, Thomas Thierauf*
Satisfiability Problems
- 96-07 *Vikraman Arvind, Jacobo Torán*
A nonadaptive NC Checker for Permutation Group Intersection
- 96-08 *David Cyrluk, Oliver Möller, Harald Rueß*
An Efficient Decision Procedure for a Theory of Fix-Sized Bitvectors with Composition and Extraction
- 96-09 *Bernd Biechele, Dietmar Ernst, Frank Houdek, Joachim Schmid, Wolfram Schulte*
Erfahrungen bei der Modellierung eingebetteter Systeme mit verschiedenen SA/RT-Ansätzen

- 96-10 *Falk Bartels, Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
Formalizing Fixed-Point Theory in PVS
- 96-11 *Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
Mechanized Semantics of Simple Imperative Programming Constructs
- 96-12 *Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
Generic Compilation Schemes for Simple Programming Constructs
- 96-13 *Klaus Achatz, Helmuth Partsch*
From Descriptive Specifications to Operational ones: A Powerful Transformation Rule, its Applications and Variants
- 97-01 *Jochen Messner*
Pattern Matching in Trace Monoids
- 97-02 *Wolfgang Lindner, Rainer Schuler*
A Small Span Theorem within P
- 97-03 *Thomas Bauer, Peter Dadam*
A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration
- 97-04 *Christian Heinlein, Peter Dadam*
Interaction Expressions - A Powerful Formalism for Describing Inter-Workflow Dependencies
- 97-05 *Vikraman Arvind, Johannes Köbler*
On Pseudorandomness and Resource-Bounded Measure
- 97-06 *Gerhard Partsch*
Punkt-zu-Punkt- und Mehrpunkt-basierende LAN-Integrationsstrategien für den digitalen Mobilfunkstandard DECT
- 97-07 *Manfred Reichert, Peter Dadam*
ADEPT_{flex} - Supporting Dynamic Changes of Workflows Without Loosing Control
- 97-08 *Hans Braxmeier, Dietmar Ernst, Andrea Mößle, Heiko Vogler*
The Project NoName - A functional programming language with its development environment
- 97-09 *Christian Heinlein*
Grundlagen von Interaktionsausdrücken
- 97-10 *Christian Heinlein*
Graphische Repräsentation von Interaktionsausdrücken
- 97-11 *Christian Heinlein*
Sprachtheoretische Semantik von Interaktionsausdrücken
- 97-12 *Gerhard Schellhorn, Wolfgang Reif*
Proving Properties of Finite Enumerations: A Problem Set for Automated Theorem Provers

- 97-13 *Dietmar Ernst, Frank Houdek, Wolfram Schulte, Thilo Schwinn*
Experimenteller Vergleich statischer und dynamischer Softwareprüfung für eingebettete Systeme
- 97-14 *Wolfgang Reif, Gerhard Schellhorn*
Theorem Proving in Large Theories
- 97-15 *Thomas Wennekers*
Asymptotik rekurrenter neuronaler Netze mit zufälligen Kopplungen
- 97-16 *Peter Dadam, Klaus Kuhn, Manfred Reichert*
Clinical Workflows - The Killer Application for Process-oriented Information Systems?
- 97-17 *Mohammad Ali Livani, Jörg Kaiser*
EDF Consensus on CAN Bus Access in Dynamic Real-Time Applications
- 97-18 *Johannes Köbler, Rainer Schuler*
Using Efficient Average-Case Algorithms to Collapse Worst-Case Complexity Classes
- 98-01 *Daniela Damm, Lutz Claes, Friedrich W. von Henke, Alexander Seitz, Adelinde Uhrmacher, Steffen Wolf*
Ein fallbasiertes System für die Interpretation von Literatur zur Knochenheilung
- 98-02 *Thomas Bauer, Peter Dadam*
Architekturen für skalierbare Workflow-Management-Systeme - Klassifikation und Analyse
- 98-03 *Marko Luther, Martin Strecker*
A guided tour through *Typelab*
- 98-04 *Heiko Neumann, Luiz Pessoa*
Visual Filling-in and Surface Property Reconstruction
- 98-05 *Ercüment Canver*
Formal Verification of a Coordinated Atomic Action Based Design
- 98-06 *Andreas Küchler*
On the Correspondence between Neural Folding Architectures and Tree Automata
- 98-07 *Heiko Neumann, Thorsten Hansen, Luiz Pessoa*
Interaction of ON and OFF Pathways for Visual Contrast Measurement
- 98-08 *Thomas Wennekers*
Synfire Graphs: From Spike Patterns to Automata of Spiking Neurons
- 98-09 *Thomas Bauer, Peter Dadam*
Variable Migration von Workflows in *ADEPT*
- 98-10 *Heiko Neumann, Wolfgang Sepp*
Recurrent V1 – V2 Interaction in Early Visual Boundary Processing
- 98-11 *Frank Houdek, Dietmar Ernst, Thilo Schwinn*
Prüfen von C-Code und Statmate/Matlab-Spezifikationen: Ein Experiment

- 98-12 *Gerhard Schellhorn*
Proving Properties of Directed Graphs: A Problem Set for Automated Theorem Provers
- 98-13 *Gerhard Schellhorn, Wolfgang Reif*
Theorems from Compiler Verification: A Problem Set for Automated Theorem Provers
- 98-14 *Mohammad Ali Livani*
SHARE: A Transparent Mechanism for Reliable Broadcast Delivery in CAN
- 98-15 *Mohammad Ali Livani, Jörg Kaiser*
Predictable Atomic Multicast in the Controller Area Network (CAN)
- 99-01 *Susanne Boll, Wolfgang Klas, Utz Westermann*
A Comparison of Multimedia Document Models Concerning Advanced Requirements
- 99-02 *Thomas Bauer, Peter Dadam*
Verteilungsmodelle für Workflow-Management-Systeme - Klassifikation und Simulation
- 99-03 *Uwe Schöning*
On the Complexity of Constraint Satisfaction
- 99-04 *Ercument Canver*
Model-Checking zur Analyse von Message Sequence Charts über Statecharts
- 99-05 *Johannes Köbler, Wolfgang Lindner, Rainer Schuler*
Derandomizing RP if Boolean Circuits are not Learnable
- 99-06 *Utz Westermann, Wolfgang Klas*
Architecture of a DataBlade Module for the Integrated Management of Multimedia Assets
- 99-07 *Peter Dadam, Manfred Reichert*
Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications. Paderborn, Germany, October 6, 1999, GI-Workshop Proceedings, Informatik '99
- 99-08 *Vikraman Arvind, Johannes Köbler*
Graph Isomorphism is Low for ZPP^{NP} and other Lowness results
- 99-09 *Thomas Bauer, Peter Dadam*
Efficient Distributed Workflow Management Based on Variable Server Assignments
- 2000-02 *Thomas Bauer, Peter Dadam*
Variable Serverzuordnungen und komplexe Bearbeiterzuordnungen im Workflow-Management-System ADEPT
- 2000-03 *Gregory Baratoff, Christian Toepfer, Heiko Neumann*
Combined space-variant maps for optical flow based navigation
- 2000-04 *Wolfgang Gehring*
Ein Rahmenwerk zur Einführung von Leistungspunktsystemen

- 2000-05 *Susanne Boll, Christian Heinlein, Wolfgang Klas, Jochen Wandel*
Intelligent Prefetching and Buffering for Interactive Streaming of MPEG Videos
- 2000-06 *Wolfgang Reif, Gerhard Schellhorn, Andreas Thums*
Fehlersuche in Formalen Spezifikationen
- 2000-07 *Gerhard Schellhorn, Wolfgang Reif (eds.)*
FM-Tools 2000: The 4th Workshop on Tools for System Design and Verification
- 2000-08 *Thomas Bauer, Manfred Reichert, Peter Dadam*
Effiziente Durchführung von Prozessmigrationen in verteilten Workflow-
Management-Systemen
- 2000-09 *Thomas Bauer, Peter Dadam*
Vermeidung von Überlastsituationen durch Replikation von Workflow-Servern in
ADEPT
- 2000-10 *Thomas Bauer, Manfred Reichert, Peter Dadam*
Adaptives und verteiltes Workflow-Management
- 2000-11 *Christian Heinlein*
Workflow and Process Synchronization with Interaction Expressions and Graphs
- 2001-01 *Hubert Hug, Rainer Schuler*
DNA-based parallel computation of simple arithmetic
- 2001-02 *Friedhelm Schwenker, Hans A. Kestler, Günther Palm*
3-D Visual Object Classification with Hierarchical Radial Basis Function Networks
- 2001-03 *Hans A. Kestler, Friedhelm Schwenker, Günther Palm*
RBF network classification of ECGs as a potential marker for sudden cardiac death
- 2001-04 *Christian Dietrich, Friedhelm Schwenker, Klaus Riede, Günther Palm*
Classification of Bioacoustic Time Series Utilizing Pulse Detection, Time and
Frequency Features and Data Fusion
- 2002-01 *Stefanie Rinderle, Manfred Reichert, Peter Dadam*
Effiziente Verträglichkeitsprüfung und automatische Migration von Workflow-
Instanzen bei der Evolution von Workflow-Schemata
- 2002-02 *Walter Guttmann*
Deriving an Applicative Heapsort Algorithm
- 2002-03 *Axel Dold, Friedrich W. von Henke, Vincent Vialard, Wolfgang Goerigk*
A Mechanically Verified Compiling Specification for a Realistic Compiler
- 2003-01 *Manfred Reichert, Stefanie Rinderle, Peter Dadam*
A Formal Framework for Workflow Type and Instance Changes Under Correctness
Checks
- 2003-02 *Stefanie Rinderle, Manfred Reichert, Peter Dadam*
Supporting Workflow Schema Evolution By Efficient Compliance Checks
- 2003-03 *Christian Heinlein*
Safely Extending Procedure Types to Allow Nested Procedures as Values

- 2003-04 *Stefanie Rinderle, Manfred Reichert, Peter Dadam*
On Dealing With Semantically Conflicting Business Process Changes.
- 2003-05 *Christian Heinlein*
Dynamic Class Methods in Java
- 2003-06 *Christian Heinlein*
Vertical, Horizontal, and Behavioural Extensibility of Software Systems
- 2003-07 *Christian Heinlein*
Safely Extending Procedure Types to Allow Nested Procedures as Values
(Corrected Version)
- 2003-08 *Changling Liu, Jörg Kaiser*
Survey of Mobile Ad Hoc Network Routing Protocols)
- 2004-01 *Thom Frühwirth, Marc Meister (eds.)*
First Workshop on Constraint Handling Rules
- 2004-02 *Christian Heinlein*
Concept and Implementation of C+++, an Extension of C++ to Support User-Defined
Operator Symbols and Control Structures
- 2004-03 *Susanne Biundo, Thom Frühwirth, Günther Palm(eds.)*
Poster Proceedings of the 27th Annual German Conference on Artificial Intelligence
- 2005-01 *Armin Wolf, Thom Frühwirth, Marc Meister (eds.)*
19th Workshop on (Constraint) Logic Programming
- 2005-02 *Wolfgang Lindner (Hg.), Universität Ulm , Christopher Wolf (Hg.) KU Leuven*
2. Krypto-Tag – Workshop über Kryptographie, Universität Ulm
- 2005-03 *Walter Guttmann, Markus Maucher*
Constrained Ordering
- 2006-01 *Stefan Sarstedt*
Model-Driven Development with ACTIVECHARTS, Tutorial
- 2006-02 *Alexander Raschke, Ramin Tavakoli Kolagari*
Ein experimenteller Vergleich zwischen einer plan-getriebenen und einer
leichtgewichtigen Entwicklungsmethode zur Spezifikation von eingebetteten
Systemen
- 2006-03 *Jens Kohlmeyer, Alexander Raschke, Ramin Tavakoli Kolagari*
Eine qualitative Untersuchung zur Produktlinien-Integration über
Organisationsgrenzen hinweg
- 2006-04 *Thorsten Liebig*
Reasoning with OWL - System Support and Insights –
- 2008-01 *H.A. Kestler, J. Messner, A. Müller, R. Schuler*
On the complexity of intersecting multiple circles for graphical display

- 2008-02 *Manfred Reichert, Peter Dadam, Martin Jurisch, Ulrich Kreher, Kevin Göser, Markus Lauer*
Architectural Design of Flexible Process Management Technology
- 2008-03 *Frank Raiser*
Semi-Automatic Generation of CHR Solvers from Global Constraint Automata
- 2008-04 *Ramin Tavakoli Kolagari, Alexander Raschke, Matthias Schneiderhan, Ian Alexander*
Entscheidungsdokumentation bei der Entwicklung innovativer Systeme für produktlinien-basierte Entwicklungsprozesse
- 2008-05 *Markus Kalb, Claudia Dittrich, Peter Dadam*
Support of Relationships Among Moving Objects on Networks
- 2008-06 *Matthias Frank, Frank Kargl, Burkhard Stiller (Hg.)*
WMAN 2008 – KuVS Fachgespräch über Mobile Ad-hoc Netzwerke
- 2008-07 *M. Maucher, U. Schöning, H.A. Kestler*
An empirical assessment of local and population based search methods with different degrees of pseudorandomness
- 2008-08 *Henning Wunderlich*
Covers have structure
- 2008-09 *Karl-Heinz Niggl, Henning Wunderlich*
Implicit characterization of FPTIME and NC revisited
- 2008-10 *Henning Wunderlich*
On span- P^{cc} and related classes in structural communication complexity
- 2008-11 *M. Maucher, U. Schöning, H.A. Kestler*
On the different notions of pseudorandomness
- 2008-12 *Henning Wunderlich*
On Toda's Theorem in structural communication complexity
- 2008-13 *Manfred Reichert, Peter Dadam*
Realizing Adaptive Process-aware Information Systems with ADEPT2
- 2009-01 *Peter Dadam, Manfred Reichert*
The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support
Challenges and Achievements
- 2009-02 *Peter Dadam, Manfred Reichert, Stefanie Rinderle-Ma, Kevin Göser, Ulrich Kreher, Martin Jurisch*
Von ADEPT zur AristaFlow[®] BPM Suite – Eine Vision wird Realität “Correctness by Construction” und flexible, robuste Ausführung von Unternehmensprozessen

2009-03

Alena Hallerbach, Thomas Bauer, Manfred Reichert
Correct Configuration of Process Variants in Provop

Ulmer Informatik-Berichte
ISSN 0939-5091

Herausgeber:
Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
89069 Ulm